FIT3143 – Parallel Computing

Assignment 1 Report

Raunak Koirala 32509987

Table of Contents

Section A: Investigate the Computational Bottleneck of a Serial Bloom Filter Algorithm for Large Datasets

a) Selecting the Bloom Filter Algorithm

For this assignment, I have chosen to investigate the Bloom Filter algorithm. The Bloom Filter is an efficient probabilistic data structure designed for membership testing, particularly in cases where false positives are acceptable. I selected this algorithm because it offers an interesting opportunity for parallelisation and can efficiently manage large datasets.

The Bloom Filter operates by utilising a bit array and multiple hash functions to store and query the presence of elements in a set. When items are inserted into the Bloom Filter, their hash values are used to set corresponding bits in the bit array. To query an item's existence, its hash values are checked in the bit array. If all corresponding bits are set, the item is considered "possibly present."

Suppose we have a large dataset of words, and we want to efficiently check whether a set of query words exists in this dataset. The traditional approach would involve comparing each query word with every word in the dataset, which can be computationally expensive.

1. Initialisation:

We start with an empty Bloom Filter, which is essentially a bit array with all bits initially set to 0.

0	0	0	0	0	0	0	0

2. Insertion:

We insert words from our dataset into the Bloom Filter using multiple hash functions. For simplicity, let's consider two hash functions (Hash Function 1 and Hash Function 2).

Word: "example"

0	0	1	0	0	1	0	0

The hash functions calculate the positions in the bit array to set to 1 based on the word "example." We repeat this process for all words in the dataset.

3. Query:

Now, when we want to check if a query word exists in our dataset, we apply the same hash functions to the query word and check the corresponding bit positions in the Bloom Filter.

Query Word: "example"

0	0	1	0	0	1	0	0

Both hash functions indicate that the corresponding bits are set to 1, suggesting that "example" may exist in the dataset. However, there's a possibility of false positives.

Advantages of the bloom filter are:

Efficiency: Bloom Filters are known for their efficiency in membership testing. They can quickly tell us whether an element is "possibly" in a set or "definitely" not in a set.

Reduced Computational Load: By using a Bloom Filter, we can significantly reduce the number of string comparisons needed, thus addressing the computational bottleneck.

By visualising the application of hash functions in the Bloom Filter, we can better understand how it reduces the computational load and provides a "possibly present" or "definitely not present" result for query words. This efficiency makes it a suitable choice for our investigation and assignment.

b) Implementing a serial version of the string-matching algorithm in C++

Technical pseudocode describing the serial implementation of the Bloom Filter using the Jenkins hash:

```
# Creating the BloomFiler
function CreateBloomFilter(size, numHashFunctions):
    bitArray = Array of booleans with 'size' elements, all set to false
    hashFunctions = 'numHashFunctions'
    return bitArray, hashFunctions

# Insert an item into the Bloom Filter
function InsertItem(bloomFilter, item):
    for i in range(hashFunctions):
        hash = JenkinsHash(item) % size(bitArray)
        Set bitArray[hash] to true

# Check if an item possibly exists in the Bloom Filter
function QueryItem(bloomFilter, item):
    for i in range(hashFunctions):
        hash = JenkinsHash(item) % size(bitArray)
        if bitArray[hash] is false:
            return false  # Item is not present
    return true  # Item is possibly present

# Main program
function Main():
    # Initialise the Bloom Filter with a bit array of size 1000 and 3 hash functions
    bloomFilter, hashFunctions = CreateBloomFilter (1000, 3)
```

```
    # Insert an item into the Bloom Filter
    InsertItem(bloomFilter, "example")

    # Check if the item possibly exists in the Bloom Filter
    exists = QueryItem(bloomFilter, "example")

    # Output the result
    if exists is true:
        Print "The item may be present in the Bloom Filter."
    else:
        Print "The item is definitely not present in the Bloom Filter."

# Start the program
Main()
```

c) Description of the word list and query list used to test the serial code

Multiple books in UTF-8 text format from https://www.gutenberg.org/ebooks/author/65 are used to test the serial code. These books include 'Little Women', 'Moby Dick' and 'The complete Work of Shakespeare'. The query file for this is for these 3 text files from ProjectGutenberg where it consists of a word per line with a number next to it. '1' indicates that it's in one or more of the text files, whereas '0' represents that it isn't present in any of them. The text files together contain more than 1.3 million words and the query file consists of over 91000 words to lookup, therefore these should be considered sufficiently large enough lists for the purposes of this test.

d) Analysis of the performance of the serial algorithm/code

Performance Analysis:

The serial code was designed to evaluate the performance of a Bloom Filter implementation, executed using a selected word list and query list, with the query list containing both individual words and sentences.

Execution Environment:

Docker Desktop: Docker Desktop was used for execution, with VM memory usage at 4GB. The CPU usage fluctuated during execution due to resource allocation within the virtual environment. These factors contributed to the computational performance of the code during testing.

Performance Metrics:

To measure the performance, the code was tested with varying bit array sizes and a constant number of hash functions. The most time-consuming portion of the code was observed to be the reading of words from text files and storing them into an array. The following metrics are a good indication of the average results that were obtained:

```
BloomFilter filter(10000000, 4);
```

```
# g++ -o assignment1 assignment1.cpp
# ./assignment1
Overall Insertion Time: 0.519461 s
Overall Query Time: 0.0327264 s
Overall Execution Time: 0.960706 s
Correct Matches: 91617
Incorrect Matches: 19
```

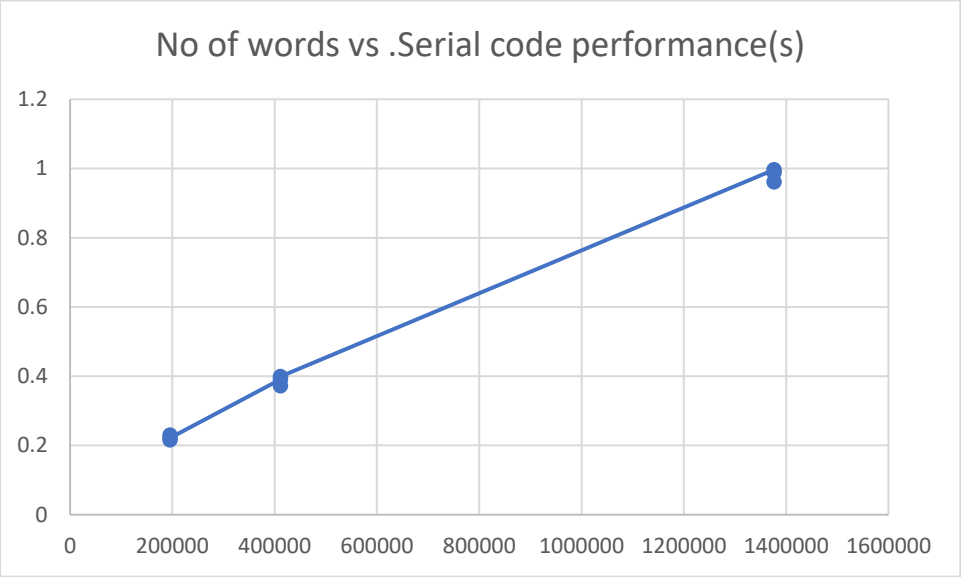
When adjusting the size of the bit array in the Bloom Filter, there was quite a big impact on accuracy. Reducing the bit array size resulted in decreased accuracy, while the execution time remained relatively stable. This is probably because of the limited capacity of the smaller bit array to represent hashed values accurately. As a result, more false positives occurred due to increased collisions among hash values. The code's execution time stayed consistent because the number of hash functions and operations remained unchanged.

Also increasing the number of hash functions in the Bloom Filter, accuracy appeared to remain similar but the execution time greatly increased. This outcome can be explained by the fact that additional hash functions require more computational resources for each item insertion and query.

As a result, the code executed more slowly, even though it maintained a similar level of accuracy in identifying item presence or absence.

The number of hash functions also seemed to play a crucial role in the performance of the Bloom Filter. Increasing the number of hash functions can improve the filter's ability to distribute items across the bit array efficiently, reducing the likelihood of false positives. However, it also increases the computational overhead during both insertion and query phases. It seemed to be a bit of a trade-off between filter accuracy and computational efficiency.

Here's a chart for number of words against the performance of the serial code



Accuracy Percentage:

On average, the code showed a very strong accuracy rate, achieving 91,617 correct matches and 19 incorrect matches out of a total of 91,636 matches. Calculating the accuracy percentage:

Accuracy Percentage = (Correct Matches / Total Matches) * 100

Accuracy Percentage = (91,617 / 91,636) * 100 ≈ 99.97%

Section B: Carry out a dependency analysis to ascertain that the string-matching algorithm is parallelizable.

a) Dependency Analysis for Parallelization with Bernstein's Conditions

In order to effectively parallelize the provided serial Bloom Filter algorithm, we need first to perform a dependency analysis while considering Bernstein's conditions. These conditions provide guidelines for determining whether the execution of multiple threads can occur simultaneously without causing errors and conflicts.

We know that Bernstein's Conditions consist of three key conditions, each addressing a specific aspect of concurrency:

1. Anti Dependency (Read-Write Conflict):

Occurs when S1 reads data that S2 writes (or vice versa).

Symbolically: $R(S1) \cap W(S2) \neq \{\}$

2. Flow Dependency (Write-Read Conflict):

Occurs when S1 writes data that S2 reads (or vice versa).

Symbolically: $W(S1) \cap R(S2) \neq \{\}$

3. Output Dependency (Write-Write Conflict):

Occurs when S1 and S2 both write to the same data.

Symbolically: $W(S1) \cap W(S2) \neq \{\}$

1. Insertion:

```
    for (const std::string& fileName : fileNames) {
        std::ifstream file(fileName);
        std::string word;

        while (file >> word) {
            allWords.push_back(word); // Store each word in the array
        }
    }

    // Insert all the words into the Bloom Filter
    for (const std::string& word : allWords) {
        filter.insert(word); // Insert each word into the Bloom Filter
    }
```

Dependency Analysis:

Anti Dependency: During the insertion phase, there are no Anti Dependencies. This means that no operations write to variables that other operations are reading. The absence of Anti Dependencies ensures that parallel execution will not violate this condition.

Flow Dependency: Flow Dependencies are present in the insertion phase as reading words from files and inserting them into the Bloom Filter is a sequential process. Each word insertion depends on the previous word being inserted. This sequential order presents a significant challenge to parallelization, as operations are inherently ordered.

Output Dependency: Output Dependencies don't exist in this phase because operations don't write to variables that other operations are writing to.

To parallelize the insertion phase effectively, we can explore the possibility of parallel file reading. By employing parallel file reading mechanisms, multiple files can be processed concurrently, eliminating Flow Dependencies.

This approach can significantly improve the overall insertion efficiency. In the insertion phase, both the allWords list and the filter Bloom Filter share memory. We can make things faster by letting different parts of the list be worked on by different threads. We would however need to be careful to avoid any mix-ups when multiple threads try to use this shared memory at the same time.

2. Query:

```
    for (const auto& query : allQueries) {
        bool exists = filter.query(query.first);
```

Dependency Analysis:

Anti Dependency: In the query phase, Anti Dependencies do not exist as there are no conflicting read and write operations.

Flow Dependency: Flow Dependencies are also minimal in the query phase, as each query is an independent operation that does not rely on previous queries.

Output Dependency: Output Dependencies do not apply here, as there are no shared variables being written to.

Although the lookup within the Bloom filter is already O(k) time complexity, the query phase does have quite significant parallelization opportunities as there isn’t many strong dependencies. We could implement where multiple queries can be executed concurrently, and parallelization can improve query performance. In the query phase, the allQueries list and the filter Bloom Filter also use shared memory. To speed things up, we can split the work of handling queries among different threads. But we have to make sure that these threads don't mess up the data when they're all working together.

Section C: Calculate the theoretical speedup of a parallel implementation of the string-matching algorithm.

a) Measure the theoretical speed up based on a fixed dataset size.

We know that Amdahl's law, s(p):

$$s_{theory}(p) = \frac{1}{r_s + \frac{r_p}{p}}$$

where rp is the parallel ratio (parallelizable portion) of the algorithm, rs is the serial ratio (non-parallelizable portion) of the algorithm and p is the number of processes (or threads).

To calculate the theoretical speedup using Amdahl's law we can use the measured values we collected before:

Time for insertion only: 0.519461 seconds.

Time for reading from files only: 0.2085726 seconds.

Time for querying only: 0.0327264 seconds.

Overall execution time: 0.960706 seconds.

Now, to calculate the percentage of time spent on each portion:

Percentage of insertion time: (0.519461 / 0.960706) * 100 ≈ 54.03%

Percentage of file reading time: (0.2085726 / 0.960706) * 100 ≈ 21.71%

Percentage of querying time: (0.0327264 / 0.960706) * 100 ≈ 3.40%

These percentages represent the portion of the code that can be parallelized. To calculate the theoretical speedup for a specific number of processes (p), we can use Amdahl's law with these percentages.

We can calculate the theoretical speedup for 4 processes:

$s(4) = 1 / [(1 - rp) + (rp / 4)]$

Where rp is the combined parallelizable portion, which is the sum of the percentages:

rp = 0.5403 (insertion) + 0.2171 (file reading) + 0.0340 (querying) = 0.7914

Now we can calculate $s(4)$:

$s(4) = 1 / [(1 - 0.7914) + (0.7914 / 4)]$

$s(4) = 1 / [(0.2086) + (0.1979)] \approx 1 / 0.4065 \approx 2.4633$

So, the theoretical speedup for 4 processes is approximately 2.4633 based on the portions that can be parallelized.

Section D: Design and develop a parallel string-matching algorithm based on data parallelism on a shared memory parallel computing architecture.

The plan is to use OpenMP to parallelize the string-matching algorithm, with a specific focus on three crucial components: parallel insertion into a Bloom filter, parallel reading of text files into an array, and parallel query processing.

The parallelizing of the insertion, reading, and querying processes is to make our string-matching algorithm work much faster and efficiently on modern computers with multicore processors. These parts of the code need parallelization more than others because they are resource-intensive tasks that can be done faster when many parts of the computer work together at the same time, where dependency isn't present, making the best use of the computer's power.

For the parallel insertion into the Bloom Filter, we will leverage OpenMP to improve execution time on the insertion process. Our approach involves creating a parallel region where multiple threads will work simultaneously to insert words. Each thread will be assigned a specific portion of the workload to ensure efficient collaboration. The insertion process will be optimised by distributing the workload evenly among threads, and each thread will perform multiple hash function evaluations in parallel for each word. This parallelization will reduce the overall insertion time, enhancing the efficiency of our string-matching algorithm.

In the case of parallel reading of text files into an array, we saw that sequential file reading can introduce substantial delays, particularly when dealing with multiple files. To resolve this issue, OpenMP is used again. The parallelization process will start by establishing a parallel region so multiple threads can work simultaneously. The #pragma omp for directive will be used to distribute the list of text files among threads, ensuring that each thread processes a portion of the files. Within each thread, words will be read from the files and collected in local vectors. These local vectors will then be merged later into a shared array.

Regarding parallel query processing, the final component of our parallel algorithm will focus on executing queries in parallel. A parallel loop will be created, allowing each thread to handle queries on the Bloom filter for a subset of the queries. The results will be aggregated using a reduction operation to add up correct and incorrect matches. This parallelization is expected to significantly reduce query time.

Technical Pseudocode:

```
# Parallel Reading of Files into an Array
function ParallelReadFilesIntoArray(fileNames, allWords):
    Parallel Region:
        For each text file in fileNames:
            Read words from the file and store them in a local vector (words)
            Combine the local vectors into 'allWords'

# Parallel Insertion into the Bloom Filter
function ParallelInsertIntoBloomFilter(bloomFilter, items):
    Parallel Region:
        For each item in items:
            Parallel InsertItem(bloomFilter, item)



# Parallel Query Processing
function ParallelQueryBloomFilter(bloomFilter, queries):
    correctMatches = 0
    incorrectMatches = 0

    Parallel Region:
        For each query in queries:
            exists = Parallel QueryItem(bloomFilter, query)
            If (exists and query.second == 1) or (!exists and query.second == 0):
                Increment correctMatches
            Else:
                Increment incorrectMatches

    Print summary of correct and incorrect matches
```

E) Analyse and evaluate the performance of the parallel algorithm

a & b) Measure the performance of the parallel algorithm/code and calculate the actual speed up

For a quick overview of an average run of the parallelised bloom filter, the insertion time and query time seem to much faster and reading from files seems a bit more efficient as well when run.

Non-parallelised results:

```
# g++ -o assignment1 assignment1.cpp
# ./assignment1
Overall Insertion Time: 0.598751 s
Overall Query Time: 0.035604 s
Overall Read Files Time: 0.406852 s
Overall Execution Time: 1.1227 s
Correct Matches: 91617
Incorrect Matches: 19
```

Parallelised results:

```
# g++ assignment1_parallel.cpp -fopenmp -o assignment1_parallel
# ./assignment1_parallel
Correct Matches: 91617
Incorrect Matches: 19
Overall Insertion Time: 0.167368 s
Overall Query Time: 0.00897756 s
Overall Read Files Time: 0.295389 s
Overall Execution Time: 0.536114 s
```

Now we'll be conducting an elaborate and thorough test by testing these algorithms against varying word sizes. We also use this data to calculate the actual speedup for both single computer and CAAS over increasing word lists. The number of threads will remain as 4 throughout for data consistency and the query list will also remain the same while changing the amount of words through the variation on the number of text files used.

Here are the results:

Serial performance on single computer

Files	Words	Overall time (s)
1	195467	0.229869
1	195467	0.216072
1	195467	0.221901
2	411191	0.390633
2	411191	0.372167
2	411191	0.399004
3	1376657	0.996668
3	1376657	0.988351
3	1376657	0.961293

Parellel performance on single computer

Files	Words	Overall time (s)
1	195467	0.150741
1	195467	0.15716
1	195467	0.156505
2	411191	0.210111
2	411191	0.214658
2	411191	0.211018
3	1,376,657	0.537928
3	1,376,657	0.547377
3	1,376,657	0.562022

Serial performance on CAAS

Files	Words	Overall time (s)
1	195467	0.214231
1	195467	0.201325
1	195467	0.232315
2	411191	0.346433
2	411191	0.320943
2	411,191	0.298544
3	1,376,657	0.693023
3	1,376,657	0.681096
3	1,376,657	0.728327

Parellel performance on CAAS

Files	Words	Overall time (s)
1	195,467	0.152332
1	195,467	0.131483
1	195,467	0.144802
2	411,191	0.219234
2	411,191	0.230148
2	411,191	0.224331
3	1,376,657	0.40576
3	1,376,657	0.401923
3	1,376,657	0.432998

	Words	Average time
1	195467	0.222614
2	411191	0.387268
3	1376657	0.982104

	Words	Average time
1	195467	0.154802
2	411191	0.211929
3	1376657	0.549109

	Words	Average time
1	195467	0.215957
2	411191	0.321973
3	1376657	0.700815

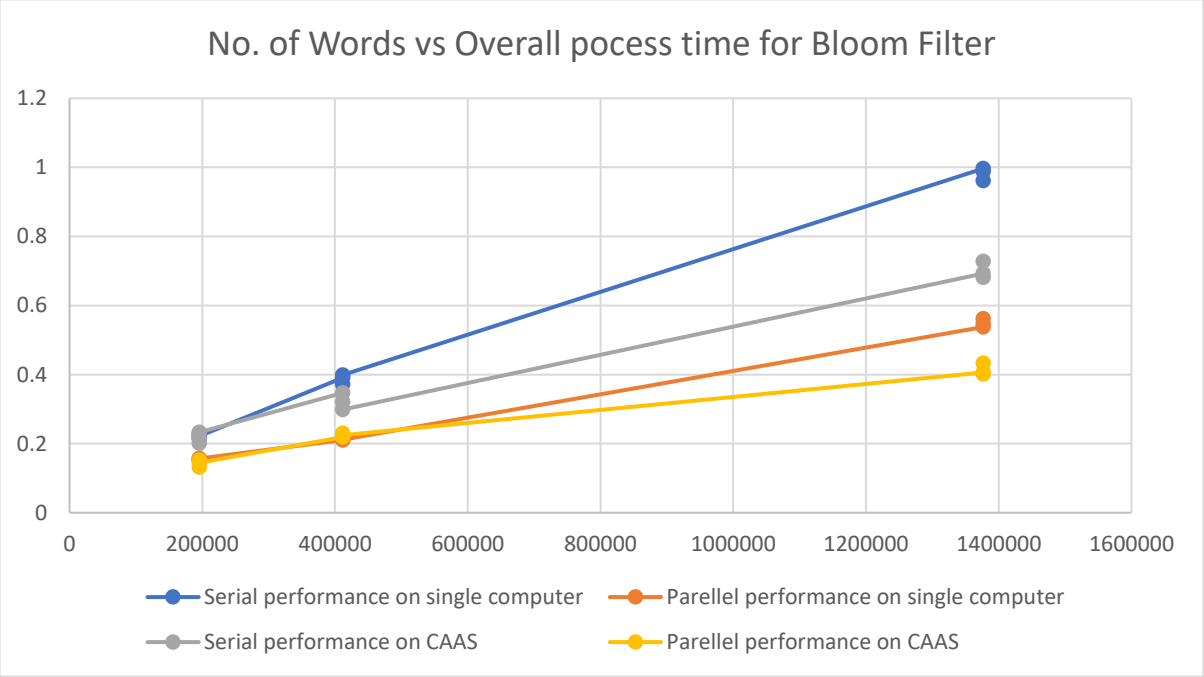
	Words	Average time
1	195467	0.142872
2	411191	0.224571
3	1376657	0.41356

Performance on single computer

Words	Serial	Parallel	Actual speed up
195467	0.222614	0.154802	1.438056
411191	0.387268	0.211929	1.827348
1376657	0.982104	0.549109	1.788541

Performance on CAAS

Words	Serial	Parallel	Actual speed up
195467	0.215957	0.142872	1.511538
411191	0.321973	0.224571	1.433726
1376657	0.700815	0.41356	1.69459



Overall, we observed that:

On a Single Computer, sublinear speedup was consistently observed across varying word counts (195,467, 411,191, and 1,376,657 words). The parallel implementation outperformed the serial version for all word counts, with actual speedup values of approximately 1.44x, 1.83x, and 1.79x, respectively. Sublinear speedup is common due to overhead and resource contention in parallel processing.

On CAAS, similar to the single computer tests, sublinear speedup was observed across different word counts on the CAAS platform. Actual speedup values of approximately 1.51x, 1.43x, and 1.69x were achieved for the respective word counts. The correlation between theoretical and actual speedup indicated that the algorithm's performance aligns with theoretical predictions.

Both the tests conducted on a single computer and the CAAS platform consistently showed sublinear speedup. This means that the performance improvement achieved by adding more processing resources and using more threads and processors, is not quite linear but is still more efficient in comparison to the serial version of the algorithm.

In the case of the serial algorithm, it utilised only a single thread or core, causing significant idle time, for example between file reads, contributing to longer execution times. The primary factor that led to this speedup in the algorithm is parallelism. By allowing multiple threads to process different segments of the input data concurrently, the overall execution time was significantly reduced. The reduction in I/O-bound operations like file reading that were causing overhead also contributed to performance improvement.

In terms of the accuracy of the parallel implementation, it demonstrated consistent accuracy, indicating that parallelisation didn’t introduce any efficiency or errors to string matching accuracy and that the parallel algorithm maintains the same level of accuracy as the serial implementation.

The theoretical speedup was calculated using Amdahl's Law, which assumed ideal conditions. In our case, the theoretical speedup was calculated to be 2.4633. Actual speedup, on the other hand, was observed during real-world execution and was influenced by various factors that Amdahl's Law did not consider. These factors included communication overhead, synchronization, and contention for shared resources. The difference between theoretical and actual speedup could be cause by these real-world complexities as the theoretical speedup assumed perfect parallelism and no overhead.

c) Optimize your algorithm to improve its performance.

Based on the test results, and the difference between the theoretical and actual speedup, within the code, particularly during word insertion into the Bloom filter and parallel reading of text files, minimising unnecessary data transfers between threads could improve efficiency. The sublinear speedup indicates that there may be room for improvement in how data is shared among threads. By optimising data access patterns and potentially using thread-local data where applicable, we could reduce communication overhead.